Abstract Specialization and its Applications

Germán Puebla
Department of Computer Science
Technical University of Madrid (UPM), Spain
german@fi.upm.es

Manuel Hermenegildo
Department of Computer Science
Technical University of Madrid (UPM), Spain, and
Dept. of C.S. and E. and C. Engineering
University of New Mexico, USA
herme@fi.upm.es

ABSTRACT

The aim of program specialization is to optimize programs by exploiting certain knowledge about the context in which the program will execute. There exist many program manipulation techniques which allow specializing the program in different ways. Among them, one of the best known techniques is *partial evaluation*, often referred to simply as program specialization, which optimizes programs by specializing them for (partially) known input data. In this work we describe *abstract specialization*, a technique whose main features are: (1) specialization is performed with respect to "abstract" values rather than "concrete" ones, and (2) *abstract interpretation* rather than standard interpretation of the program is used in order to propagate information about execution states. The concept of abstract specialization is at the heart of the specialization system in `CiaoPP`, the `Ciao` system preprocessor. In this paper we present a unifying view of the different specialization techniques used in `CiaoPP` and discuss their potential applications by means of examples. The applications discussed include program parallelization, optimization of dynamic scheduling (concurrency), and integration of partial evaluation techniques.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming—*Automatic analysis of algorithms, Program transformation*; D.3.2 [**Programming Languages**]: Language classification—*Constraint and logic languages*; D.3.4 [**Programming Languages**]: Processors—*Optimization, Compilers*

General Terms

Languages, Performance, Theory

Keywords

Abstract Interpretation, Program Specialization, Partial Evaluation, Program Optimization, Program Parallelization, Logic Programming, Static Analysis

1. INTRODUCTION

The aim of program optimization is, given a program P to obtain another program P' which is semantically equivalent to P but behaves better for some criteria of interest. One typical way of optimizing programs is by *specializing* them for some particular context. This allows automatically overcoming losses in performance which are due to general purpose algorithms. This situation is becoming more and more frequent due to the use of techniques such as reuse of general-purpose programs and libraries, and software components, which facilitate development but can result in large programs and even waste of computing resources. More precisely, the aim of program specialization is, given a program P and certain knowledge ϕ about the context in which P will be executed, to obtain a program P_ϕ which is equivalent to P for all contexts which satisfy ϕ and which behaves better from some given point of view.

In the case of *partial evaluation* [4, 24], the knowledge ϕ which is exploited is the so-called *static* data, which corresponds to (partial) knowledge at specialization (compile) time about the input data. Data which is not known at specialization time is called *dynamic*. The program is optimized by performing at specialization time those parts of the program execution which only depend on static data.

Another very general setting for specialization specially relevant in the context of logic programs, which has been proposed in [39], is to define the knowledge about the context as a so-called *static property* $\phi(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n$ are the formal arguments of the top-level procedure P and ϕ is defined as a logic program. However, this approach suffers from an important difficulty in using the context information in an automated and effective way.

The approach we follow in *abstract specialization* is that the information ϕ available on the context is captured by an *abstract substitution*. One advantage of this approach is that there are well known techniques which allow handling information represented as abstract substitutions by using *abstract interpretation* techniques [6].

1.1 An Overview of Specialization Techniques

For the purpose of comparing different existing techniques, let us classify the existing specialization techniques according to how the final, optimized, program is obtained. Of course, this classification is rather crude and many of the existing techniques can be seen as a combination of the three approaches which we will discuss. The first approach which we describe, and which we will call *program with annotations*, consists of two phases. During the first phase,

some *static program analysis* technique is used in order to annotate the program with analysis information. In the second phase, the program is optimized using the information obtained. This approach is conceptually simple, though either or both of the phases mentioned can indeed be rather complex. A well known example of this kind of techniques is the "off-line" approach to partial evaluation, in which a *binding-time analysis* phase is followed by another one in which the residual program is generated.

The second class of techniques we consider, which we will call the *transformational* approach, is based on program transformation techniques, such as fold/unfold transformations (such as the ones developed in [3, 48]). In this scheme, a series of n semantic-preserving program-transformation steps are performed such that initially $P = P_0$. Then, each P_{i+1} is obtained from P_i by applying some transformation T_i, i.e., $P_{i+1} = T_i(P_i)$, which preserves the semantics of the program. Finally $P' = P_n$. Transformational techniques are very powerful, the main difficulty being in automatically deciding a proper sequence of programs transformations to perform in order to obtain (an optimal) program P'.

The third and last possibility which we consider, and which we will denote the *semantic* approach, is based on the existence of an algorithm $\mathcal{S}$ which, given a program P and some knowledge ϕ, builds a semantic representation of the program $\mathcal{S}(P, \phi)$ which captures the behaviour of P in some precise way for all contexts which satisfy ϕ. Then, there is a code generation algorithm which builds the program P_ϕ from $\mathcal{S}(P, \phi)$ in a straightforward way. Often this semantic representation can be seen as a graph. The kind of graph obtained depends on the particular semantics used by the algorithm. The "on-line" approach to partial evaluation is, in our terminology, a semantic approach since the behaviour of the program is precisely captured by the partial evaluation algorithm.

A particular algorithm for the on-line partial evaluation of logic programs is *partial deduction* [35, 26]. Though on-line partial evaluation can be considered an instance of fold/unfold transformations, the comparatively significant success of partial deduction techniques is probably due to the fact that they are often formalized as a semantic approach. I.e., an algorithm exists which can be used to build the semantic representation of the program. The existing algorithms for partial deduction [35, 10, 28] are parameterized by different control strategies. Usually, control is divided into components: "local control," which controls the unfolding for a given atom, and "global control," which ensures that the set of atoms for which a partial evaluation is to be computed remains finite. Several strategies for global and local control have been proposed which produce good-quality partial evaluations of programs [36, 31]. Regarding the correctness of partial deduction, two conditions, defined on the set of atoms to be partially evaluated, have been identified which ensure correctness of the transformation: "closedness" and "independence" [35].

1.2 Abstract Specialization through A Motivating Example

One of the distinguishing features of logic programming (LP) is that arguments to procedures can be uninstantiated variables. This, together with the search execution mechanism available (generally backtracking) makes it possible to have *multi-directional* procedures. I.e., rather than having fixed input and output arguments, execution can be "reversed". Thus, we may compute the "input" arguments from known "output" arguments.

EXAMPLE 1.1. *Consider the logic program below. As usual in LP,* predicates *(procedures) are referred to in the text as* `name/arity`*, where arity is the number of arguments of the predicate. The predicate* `ground/1` *is a boolean test which succeeds if and only if its argument is bound at run-time to a term without variables, and the predicate* `is/2` *(used as an infix binary operator) computes the arithmetic value of its second (right) argument and unifies it with its first (left) argument.*

```
plus(X,Y,Z):- ground(X),ground(Y),!,Z is X + Y.
plus(X,Y,Z):- ground(Y),ground(Z),!,X is Z - Y.
plus(X,Y,Z):- ground(X),ground(Z),!,Y is Z - X.
```

The procedure `plus/3` *defines the relation such that the third argument is the addition of the first and second arguments. The procedure* `plus/3` *is multi-directional. For example, the call* `plus(1, 2, Sum)` *can be used to compute the addition of 1 and 2. Also, the call* `plus(Num, 2, 3)` *can be used to determine which is the number* `Num` *such that when added to 2 returns 3.*

Thus, the definition of `plus/2` behaves *declaratively* as long as at least two of the input arguments are ground. However, this good behavior of `plus/3` when compared to a mono-directional operation such as `is/2` is at the expense of some overhead which is incurred at run-time in order to select the appropriate clause to execute out of the three existing ones. Imagine now that at compile-time it is known that the call to `plus/3` will be of the form `plus(1, 2, Sum)`. In such case it is clear that the first clause will be selected and the execution will return the value 3 for the argument `Sum`. This is a typical example of an execution which can benefit from (traditional, "concrete") partial evaluation where ϕ is the knowledge that the initial call is `plus(X,Y,Z)` with `X=1` and `Y=2`.

In spite of the relative maturity of partial evaluation of logic programs, it is well known that the technique has certain shortcomings. Imagine we are interested in optimizing the code:

```
p(X,Y,Res):- plus(X,Y,Tmp), plus(1,Tmp,Res).
```

where `plus/3` is defined as above. By observing the program we can conclude that after the execution of the call `plus(X,Y,Tmp)` all three arguments are ground. As a result, the call `plus(1,Tmp,Res)` can be optimized to `Res is 1 + Tmp`.

Unfortunately, in traditional partial evaluation no information on the value of the argument `Tmp` is propagated to the call `plus(1,Tmp,Res)`. The intrinsic problem underlying this shortcoming of partial evaluation is that the only information which can be captured about values of arguments are *concrete* values. In the case of logic programming, values are captured by *substitutions*. This shortcoming of partial evaluation has been identified and several proposals exist which try to overcome it. Our proposal, *abstract specialization*, addresses this problem directly. Abstract specialization allows specializing calls with respect to *abstract substitutions* instead of *concrete substitutions* as in traditional partial evaluation. As will be discussed in Section 2, abstract substitutions are in this context finite representations of possibly infinite sets of data. Each such representation method is called an *abstract domain*. The kind of informa-

tion which can be captured by abstract substitutions varies from one abstract domain to another. For example, we can have an abstract domain which allows capturing type information.[1] Such domain can be used to determine that in the call `plus(1,Tmp,Res)` the argument `Tmp` is bound to a number. We can use this information in order to *abstractly execute* the two ground terms in the first clause of `plus/3` to the value *true*. We can even execute the `!/0` procedure call and eliminate the rest of clauses for `plus/3`[2]. We can thus optimize the original program to:

```
p(X,Y,Res):- plus(X,Y,Tmp), Res is 1 + Tmp.
```

Also, the call `plus(X,Y,Tmp)` can be optimized. Since `Tmp` is a variable which is local to the clause, it can be determined to be a free variable (and thus definitely not ground). Thus, the program can be optimized to:

```
p(X,Y,Res):- plus1(X,Y,Tmp), Res is 1 + Tmp.
plus1(X,Y,Z):- ground(X),ground(Y),!,Z is X + Y.
```

where `plus1/3` is a specialized version of `plus/3`. Generalizing from the examples above we can develop a specialization system which is able to perform the optimizations shown above. The specialization system will be able to: (1) capture more general information than traditional substitutions, i.e., it will capture abstract substitutions, (2) propagate such information in a correct way using a suitable semantics, and (3) carry out the optimizations enabled by the information available.

In the rest of the paper we present such a system. The structure of the paper is as follows. After recalling some basic concepts of abstract interpretation in Section 2, we present the concept of abstract executability in Section 3. Then we introduce our generic abstract multiple specialization framework in Section 4. We continue with several applications of abstract multiple specialization: in the context of automatic program parallelization in Section 5, optimization of dynamic scheduling in Section 6, and partial evaluation in Section 7. Finally, Section 8 discusses related work and Section 9 presents some conclusions.

2. ABSTRACT INTERPRETATION

Static Program analysis aims at deriving at compile-time certain properties of the run-time behavior of a program. We provide some background and notation on abstract interpretation [6], which is arguably one of the most successful techniques for static program analysis.

In abstract interpretation, the execution of the program is "simulated" on an *abstract domain* (D_α) which is simpler than the actual, *concrete domain* (D). An abstract value is a finite representation of a, possibly infinite, set of actual values in the concrete domain (D). The set of all possible abstract semantic values represents an abstract domain D_α which is usually a complete lattice or cpo which is ascending chain finite. However, for this study, abstract interpretation is restricted to complete lattices over sets both for the concrete $\langle 2^D, \subseteq\rangle$ and abstract $\langle D_\alpha, \sqsubseteq\rangle$ domains.

Abstract values and sets of concrete values are related via a pair of monotonic mappings $\langle\alpha,\gamma\rangle$: *abstraction* $\alpha : 2^D \rightarrow D_\alpha$, and *concretization* $\gamma : D_\alpha \rightarrow 2^D$, such that

$$\forall x \in 2^D :\ \gamma(\alpha(x)) \supseteq x \quad \text{and} \quad \forall y \in D_\alpha :\ \alpha(\gamma(y)) = y.$$

Note that in general $\sqsubseteq$ is induced by $\subseteq$ and α (in such a way that $\forall\lambda,\lambda' \in D_\alpha :\ \lambda \sqsubseteq \lambda' \Leftrightarrow \gamma(\lambda) \subseteq \gamma(\lambda')$). Similarly, the operations of *least upper bound* ($\sqcup$) and *greatest lower bound* ($\sqcap$) mimic those of 2^D in some precise sense.

EXAMPLE 2.1 (A DOMAIN FOR MODE ANALYSIS). *Consider the following toy abstract domain D_α which captures* mode *information (i.e., the state of instantiation of program variables upon procedure call). An abstract substitution λ over a set of variables $\overline{X} = \{X_1,\ldots,X_n\}$ assigns to each variable X_i a value v in the set $\{ground, var, any\}$ where each v represents an infinite set of terms. The fact that a variable X_i is assigned an abstract value v indicates that X_i will be bound at run-time to some term belonging to v. ground is the set of all terms without variables; var is the set of unbound variables (possibly aliased to other unbound variables); and any is the set of all terms. The abstract domain is complemented by the abstract substitutions $\bot$ and $\top$. As usual in abstract interpretation, $\bot$ denotes the abstract substitution such that $\gamma(\bot) = \emptyset$. The substitution $\top$ is such that $\gamma(\top) = D$. In our domain, $\top$ corresponds to assigning any to each variable in $\overline{X}$.* □

Since our discussion will concentrate on logic programs, we also recall some classical definitions in logic programming. An *atom* has the form $p(t_1,...,t_n)$ where p is a predicate symbol and the t_i are terms. We often use $\overline{t}$ to denote a tuple of terms. A *clause* is of the form H :- $B_1,\ldots,B_n$ where H, the *head*, is an atom and $B_1,\ldots,B_n$, the *body*, is a possibly empty finite conjunction of atoms. Atoms in the body of a clause are often called *literals*. A *program* is a finite sequence of clauses.

2.1 Goal-Dependent analysis

Goal-dependent analyses are characterized by generating information which is valid only for a restricted set of calls to a predicate, as opposed to goal-independent analyses whose results are valid for any call to the predicate. Goal-dependent analyses allow obtaining results which are *specialized* (restricted) to a given context. As a result, they provide in general better (stronger) results than goal-independent analyses. In addition, goal-dependent analyses provide information on both the call and success states for each predicate, whereas goal-independent analyses in principle only provide information on success states of predicates. For these reasons, and since program specialization greatly relies on information about call states to predicates, we will restrict the discussion to goal-dependent analyses.

In order to improve the accuracy of goal-dependent analyses, some kind of description of the *initial* calls to the program should be given.[3] With this aim, we will use `entry` declarations in the spirit of [2]. Their role is to restrict the starting points of analysis to only those calls which satisfy a declaration of the form '`:- entry` *Pred* : *Call.* ' where *Call* is an abstract call substitution for *Pred*. For example, the following declaration informs the analyzer that at run-time all initial calls to the predicate `qsort/2` will have a term without variables in the first argument position:

```
:- entry qsort(A,B) : ground(A).
```

[1] Alternatively we could use an abstract domain which captures groundness information natively and obtain the same optimized program.

[2] The procedure call `!/0` is used to eliminate other alternatives.

[3] Predicate calls which are not initial will be called *internal*.

Property	Definition	Sufficient condition
L is abstractly executable to *true* in P	$RT(L,P) \subseteq TS(L,P)$	$\exists \lambda' \in A_{TS}(\overline{B}, D_\alpha) : \lambda_L \sqsubseteq \lambda'$
L is abstractly executable to *false* in P	$RT(L,P) \subseteq FF(L,P)$	$\exists \lambda' \in A_{FF}(\overline{B}, D_\alpha) : \lambda_L \sqsubseteq \lambda'$

Table 1: Abstract Executability

Though our framework allows having several entry declarations (for the same or different exported predicates), for the sake of clarity of the presentation we restrict ourselves to having one entry declaration only. Also, CiaoPP [21] supports a more general language, which includes properties defined in the source language [40]. In this setting, goal dependent abstract interpretation takes as input (1) a program P (2) an atom p, (3) an abstract substitution λ in (4) an abstract domain D_α which describes restrictions on the initial values, and computes a set of triples $Analysis(P, p, \lambda, D_\alpha) = \{\langle p_1, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle p_n, \lambda_n^c, \lambda_n^s\rangle\}$. In each triple $\langle p_i, \lambda_i^c, \lambda_i^s\rangle$, p_i is an atom and λ_i^c and λ_i^s are, respectively, the abstract call and success substitutions.[4] Due to space limitations, and given that it is now well understood, we do not describe here how we compute $Analysis(P, p, \lambda, D_\alpha)$. More details can be found in [22, 44] and their references. Given $Analysis(P, p, \lambda, D_\alpha) = \{\langle p_1, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle p_n, \lambda_n^c, \lambda_n^s\rangle\}$, correctness of abstract interpretation guarantees that the following propositions hold:

PROPOSITION 2.2 (CORRECTNESS W.R.T. SUCCESSES). *The abstract success substitutions cover all the concrete success substitutions which appear during execution, i.e., $\forall i = 1..n\ \forall \theta_c \in \gamma(\lambda_i^c)$ if $p_i\theta_c$ succeeds in P with computed answer substitution θ_s then $\theta_s \in \gamma(\lambda_i^s)$.*

PROPOSITION 2.3 (CORRECTNESS W.R.T. CALLS). *The abstract call substitutions cover all the concrete calls which appear during executions described by $\langle p, \lambda\rangle$. I.e., for any concrete call c originated from an initial goal $p\theta$ s.t. $\theta \in \gamma(\lambda)$: $\exists \langle p_j, \lambda_j^c, \lambda_j^s\rangle \in Analysis(P, p, \lambda, D_\alpha)$ s.t. $c = p_j\theta'$ and $\theta' \in \gamma(\lambda_j^c)$.*

Proposition 2.3 is related to the closedness condition [35] required in partial deduction. A tuple $\langle p_j, \lambda_j^c, \bot\rangle$ indicates that all calls to predicate p_j with substitution $\theta \in \gamma(\lambda_j^c)$ either fail or loop, i.e., they do not produce any success substitutions. An analysis is said to be *multivariant* if more than one triple $\langle p, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle p, \lambda_n^c, \lambda_n^s\rangle$ $n > 1$ with $\lambda_i^c \neq \lambda_j^c$ for some i, j may be computed for the same predicate p.

3. ABSTRACT EXECUTABILITY

The concept of *abstract executability* [17, 45] allows reducing at compile-time certain program fragments to the values *true*, *false*, or *error*, or to a simpler program fragment, by application of the information obtained via abstract interpretation. This allows optimizing and transforming the program (and also detecting errors at compile-time in the case of *error*).

[4]Actually, the analyzers used in practice generate information not only at the *predicate level*, as stated here for simplicity, but also at the *clause literal* level.

For simplicity, we will limit herein the discussion to reducing a procedure call or program fragment L (for example, a "literal" in the case of logic programming) to either *true* or *false*. Each run-time invocation of the procedure call L will have a *local environment* which stores the particular values of each variable in L for that invocation. We will use θ to denote this environment (composed of assignments of values to variables, i.e., substitutions) and the restriction (projection) of the environment θ to the variables of a procedure call L is denoted $\theta|_L$.

We now introduce some definitions. Given a procedure call L to a predicate which performs no side-effects in a program P we define the *trivial success set* of L in P as $TS(L,P) = \{\theta|_L : L\theta$ succeeds exactly once in P with empty answer substitution $(\epsilon)\}$. Similarly, given a procedure call L from a program P we define the *finite failure set* of L in P as $FF(L,P) = \{\theta|_L : L\theta$ fails finitely in $P\}$.

Finally, given a procedure call L from a program P we define the *run-time substitution set* of L in P, denoted $RT(L,P)$, as the set of all possible substitutions (run-time environments) in the execution state just prior to executing L in any possible execution of program P.

Table 1 shows the conditions under which a procedure call L is abstractly executable to either *true* or *false*. In spite of the simplicity of the concepts, these definitions are in general not directly applicable in practice since $RT(L,P)$, $TS(L,P)$, and $FF(L,P)$ are generally not known at compile time. However, a *collecting semantics* is generally used as concrete semantics for abstract interpretation so that analysis computes for each procedure call L in the program an abstract substitution λ_L which is a safe approximation of $RT(L,P)$, i.e. $\forall L \in P\ .\ RT(L,P) \subseteq \gamma(\lambda_L)$.

Also, under certain conditions we can compute either automatically or by hand sets of abstract values $A_{TS}(\overline{L}, D_\alpha)$ and $A_{FF}(\overline{L}, D_\alpha)$ where $\overline{L}$ stands for the *base form* of L, i.e., all the arguments of L contain distinct free variables. Intuitively, they contain abstract values in domain D_α which guarantee that the execution of $\overline{L}$ trivially succeeds (resp. finitely fails). Soundness requires that $\forall \lambda \in A_{TS}(\overline{L}, D_\alpha)$ $\gamma(\lambda) \subseteq TS(\overline{L}, P)$ and $\forall \lambda \in A_{FF}(\overline{L}, D_\alpha)$ $\gamma(\lambda) \subseteq FF(\overline{L}, P)$.

Even though the simple optimizations illustrated above may seem of narrow applicability, in fact for many builtin procedures such as those that check basic types or which inspect the structure of data, even these simple optimizations are indeed very relevant. Two non-trivial examples are their application to simplifying independence tests in program parallelization [45], discussed in Section 5, and the optimization of delay conditions in logic programs with dynamic procedure call scheduling order [41], discussed in Section 6.

Also, the class of optimizations which can be performed can be made to cover traditional lower-level optimizations as well, provided the lower-level code to be optimized is

"reflected" (i.e., is made explicit) at the source level or if the abstract interpretation is performed directly at the object level.

4. ABSTRACT MULTIPLE SPECIALIZATION

The traditional approach used in analysis-based optimizing compilers is to first analyze the program and then use the information in $Analysis(P, p, \lambda, D_\alpha)$ to annotate the program with information which is then used for optimization. Often, the underlying analysis algorithm is multi-variant. However, analysis information for the different versions of a procedure call is "flattened", i.e., "lubbed" together before being used for optimization. Though this approach allows important optimizations, it produces optimizations which may be suboptimal when compared with the optimizations which could be achieved if separate specializations were implemented for the different versions considered by multi-variant analysis. More precisely, suppose $\{\langle p_j, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle p_j, \lambda_n^c, \lambda_n^s\rangle\}$ $n > 1$ are the tuples in $Analysis(P, p, \lambda, D_\alpha)$ for predicate p_j. Generally, only one version for p_j is implemented, which is equivalent to specializing p_j w.r.t. $\lambda_1^c \sqcup \lambda_2^c, \ldots \sqcup \lambda_n^c$.

The main idea that we will exploit is to generate a different version of p_j for each tuple $\langle p_j, \lambda_i^c, \lambda_i^s\rangle$. Then, each version can be specialized w.r.t. λ_i^c regardless of the rest of the call substitutions λ_j^c $\forall j \neq i$. Hopefully, this will lead to further opportunities for optimization in each particular version. Note that if analysis terminates the number of tuples in $Analysis(P, p, \lambda, D_\alpha)$ for each predicate must be finite, and thus the resulting program will be finite. We will refer to this kind of specialization as *abstract multiple specialization* [43, 45]. An important observation here is that abstract multiple specialization is not a *program with annotations* approach but rather a *semantic* approach in the terminology of Section 1.1.

4.1 Analysis And–Or Graphs

Traditional, goal dependent abstract interpreters for LP based on Bruynooghe's analysis framework [1], in order to compute $Analysis(P, p, \lambda, D_\alpha)$, construct an and–or graph which corresponds to (or approximates) the abstract semantics of the program. We will denote by $AO(P, p, \lambda, D_\alpha)$ the and–or graph computed by the analyzer for a program P with calling pattern $\langle p, \lambda\rangle$ using the domain D_α. Such and–or graph can be viewed as a finite representation of the (possibly infinite) set of and–or trees explored by the (possibly infinite) concrete execution. Concrete and–or trees which are infinite can be represented finitely through a widening into a rational tree. Also, the use of abstract values instead of concrete ones allows representing infinitely many concrete execution trees with a single abstract analysis graph. Finiteness of $AO(P, p, \lambda, D_\alpha)$ (and thus termination of analysis) is achieved by considering an abstract domain D_α with certain characteristics (such as being finite, or of finite height, or without infinite ascending chains) or by the use of a *widening* operator [6].

The graph has two sorts of nodes: those which correspond to atoms (called *or–nodes*) and those which correspond to clauses (called *and–nodes*). Or–nodes are triples $\langle p_i, \lambda_i^c, \lambda_i^s\rangle$. As before, λ_i^c and λ_i^s are, respectively, a pair of abstract call and success substitutions for the atom p_i. For clarity, in the figures the atom p_i is superscripted with λ^c to the left and λ^s to the right of p_i respectively. For example, the or–node $\langle p(A), \{\}, \{A/a\}\rangle$ is depicted in the figure as ${}^{\{\}}p(A)^{\{A/a\}}$. And–nodes are pairs $\langle Id, H\rangle$ where Id is a unique identifier for the node and H is the head of the clause to which the node corresponds. In the figures, they are represented as triangles and H is depicted to the right of the triangles. Note that the substitutions (atoms) labeling and–nodes are concrete whereas the substitutions labeling or–nodes are abstract. Finally, squares are used to represent the empty (true) atom. Or–nodes have arcs to and–nodes which represent the clauses with which the atom (possibly) unifies. And–nodes have arcs to or–nodes which represent the atoms in the body of the clause. Note that several instances of the same clause may exist in the analysis graph of a program. In order to avoid conflicts with variable names, clauses are standardized apart before adding to the analysis graph the nodes which correspond to such clause.

Intuitively, analysis algorithms are just graph traversal algorithms which, given P, p, λ, and D_α, build $AO(P, p, \lambda, D_\alpha)$ by processing program clauses from left to right, adding the required nodes, and computing success substitutions until a global fixpoint is reached. For a given P, p, λ, and D_α there may be many different analysis graphs. However, there is a unique *least analysis graph* which gives the most precise information possible. This analysis graph corresponds to the least fixpoint of the abstract semantic equations. Each time the analysis algorithm creates a new or–node for some p_i and λ_i^c and before computing the corresponding λ_i^s, it checks whether $Analysis(P, p, \lambda, D_\alpha)$ already contains a tuple for (a variant of) p_i and λ_i^c. If that is the case, the or–node is not expanded and the already computed λ_i^s stored in $Analysis(P, p, \lambda, D_\alpha)$ is used for that or–node. This is done both for efficiency and for avoiding infinite loops when analyzing recursive predicates. As a result, several instances of the same or–node may appear in AO, but only one of them is expanded. We denote by $\mathsf{expansion}(N)$ the instance of the or–node N which is expanded. If there is no tuple for p_i and λ_i^c in $Analysis(P, p, \lambda, D_\alpha)$, the or–node is expanded, λ_i^s computed, and $\langle p_i, \lambda_i^c, \lambda_i^s\rangle$ added to $Analysis(P, p, \lambda, D_\alpha)$. Note that the success substitutions λ_i^s stored in $Analysis(P, p, \lambda, D_\alpha)$ are tentative and may be updated during analysis. Only when a global fixpoint is reached the success substitutions are safe approximations of the concrete success substitutions.

For clarity of the presentation, in the examples below we use the concrete domain as abstract domain. However, this cannot be done in general since analysis may not terminate. We will present other examples with more realistic domains later in the paper.

EXAMPLE 4.2. *Consider the simple example program below taken from [28]. Figure 1 depicts a possible result of analysis for the initial call $p(A)$ with A unrestricted. The dotted arc indicates that the corresponding or–nodes have renamings of the same abstract call substitution.*

```
p(X):- q(X), r(X).
q(a).
q(X):- q(X).
r(a).
r(b).
```

Clearly, in the example program above the clause `r(b)` is useless and could be eliminated. Note that analysis has

ALGORITHM 4.1 (CODE GENERATION). *Given* $Analysis(P, p, \lambda, D_\alpha)$ *and* $AO(P, p, \lambda, D_\alpha)$ *generated by analysis for a program* P *an atom* p *with abstract substitution* $\lambda \in D_\alpha$ *do:*

- *For each tuple* $N = \langle a(\overline{t}), \lambda^c, \lambda^s \rangle \in Analysis(P, p, \lambda, D_\alpha)$ *generate a distinct predicate with name* $pred_N = \mathsf{name}(\langle a(\overline{t}), \lambda^c, \lambda^s \rangle)$.
- *Each predicate* $pred_N$ *is defined by the sequence of clauses*
  - $(pred_N(\overline{t}_1)$:- $b'_1) :: \ldots :: (pred_N(\overline{t}_n)$:- $b'_n)$
    where $\mathsf{expansion}(N, AO) = O_N$ *and*
    $\mathsf{children}(O_N, AO) = \langle Id_1, p_1(\overline{t}_1) \rangle :: \ldots :: \langle Id_i, p_i(\overline{t}_i) \rangle :: \ldots :: \langle Id_n, p(\overline{t}_n) \rangle$
- *Each body* b'_i *is defined as*
  - $b'_i = (pred_{i1}(\overline{t}_{i1}), \ldots, pred_{ik_i}(\overline{t}_{ik_i}))$
    where $pred_{ij} = \mathsf{name}(\langle a_{ij}(\overline{t}_{ij}), \lambda^c_{ij}, \lambda^s_{ij} \rangle)$, *and*
    $\mathsf{children}(\langle Id_i, p_i(\overline{t}_i) \rangle, AO) = \langle a_{i1}(\overline{t}_{i1}), \lambda^c_{i1}, \lambda^s_{i1} \rangle :: \ldots :: \langle a_{ik_i}(\overline{t}_{ik_i}), \lambda^c_{ik_i}, \lambda^s_{ik_i} \rangle$.

Figure 2: Algorithm for Code Generation

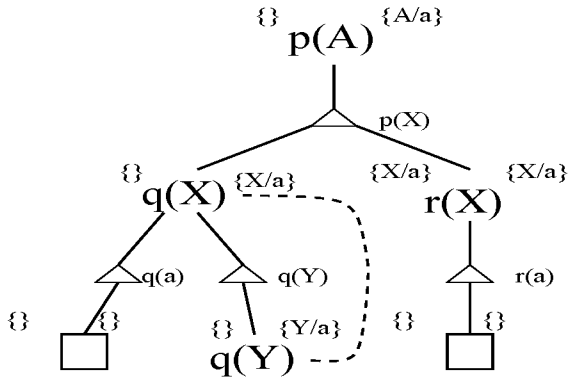

Figure 1: And–or analysis graph for a recursive program

determined that in all successes of `q(X)`, and thus in calls to `r(X)`, the argument `X` will be bound to the value `a`. This is achieved by performing a fixpoint computation on the success values of `q(X)`. This is why in Figure 1 the or–node $\langle r(X), \{X/a\}, \{X/a\} \rangle$ only has one child (and–node).

4.2 Code Generation from an And–Or Graph

After introducing some notation, Algorithm 4.1 which generates a program from an analysis and–or graph is presented in Figure 2. Given a non-root node N, we denote by $parent(N, AO)$ the node $M \in AO$ such that there is an arc from M to N in AO, and $\mathsf{children}(N, AO)$ is the sequence of nodes $N_1 :: \ldots :: N_n$ $n \geq 0$ such that there is an arc from N to N' in AO iff $N' = N_i$ for some i and $\forall i, j = 0, \ldots, n$. N_i is to the left of N_j in AO iff $i < j$. Note that $\mathsf{children}(N, AO)$ may be applied both to or– and and–nodes. We assume the existence of an injective function name which (1) given $Analysis(P, p, \lambda, D_\alpha)$ returns a unique predicate name for each tuple and (2) $\mathsf{name}(\langle q(\overline{t}), \lambda^c, \lambda^s \rangle) = q$ iff $q(\overline{t}) = p$ (the exported predicate) and $\lambda^c = \lambda$ (the restriction on initial calls), to ensure that top-level – exported – predicate names are preserved.

Basically, the algorithm for code generation shown in Figure 2 creates a different version (predicate) name for each different (abstract) call substitution λ^c to each predicate p_i in the original program. This is easily done by associating a version to each or–node. Note that in principle such versions are identical except that atoms in clause bodies are renamed to always call the appropriate version. Let $AO(P, p, \lambda, D_\alpha)$ be an and–or graph. We denote by $P' = code\_gen(AO(P, p, \lambda, D_\alpha))$ that P' is the program obtained by applying Algorithm 4.1 to $AO(P, p, \lambda, D_\alpha)$. Correctness of P' w.r.t. P for all initial calls $p\theta$ with $\theta \in \gamma(\lambda)$ is given by the correctness of the abstract interpretation procedure, as the extended program P' is obtained by simply materializing the (implicit) program with multiple versions from which the analysis has obtained its information.

EXAMPLE 4.3. *The program generated by the code generation algorithm for the and–or graph in Figure 1 is shown below. The useless clause* `r(b)` *has been eliminated.*

```
p(X):- q(X), r(X).
q(a).
q(X):- q(X).
r(a).
```

The example above shows how the use of and–or graphs allows removing useless clauses. The example below shows how generating multiple specialized versions of a predicate can lead to optimizations which are not possible if only one version were implemented.

EXAMPLE 4.4. *Consider again the program* P *in Example 1.1. The and–or graph* $AO(P, p, \top, D_\alpha)$ *where* D_α *is a domain which captures mode information will have two or-nodes for predicate* `plus/3` *with different abstract call substitutions (we abbreviate ground by g):*
$\langle plus(X', Y', Z'), \{Z'/var\}, \{X'/g, Y'/g, Z'/g\} \rangle$ *and* $\langle plus(X'', Y'', Z''), \{X''/g, Y''/g\}, \{X''/g, Y''/g, Z''/g\} \rangle$.
Now each of these call patterns can be optimized separately by abstractly executing the groundness tests. The final specialized program obtained is shown below:

```
p(X,Y,Res) :- plus1(X,Y,Tmp), plus2(1,Tmp,Res).
plus1(X,Y,Z) :- ground(X), ground(Y), !, Z is X+Y.
plus2(X,Y,Z) :- Z is X+Y.
```

Note that this program could be further improved by unfolding the call `plus2(1,Tmp,Res)`. This will be further discussed in Section 7. Also, two versions have been generated

for predicate `plus/3`, namely `plus1/3` and `plus2/3`. In order to avoid code explosion our system performs a minimizing step a posteriori on the and–or graph in order to produce the minimal number of versions while maintaining all optimizations [45].

5. PROGRAM PARALLELIZATION

The final aim of parallelism is to achieve the maximum speed (effectiveness) while computing the same solution (correctness) as the sequential execution. The two main types of parallelism which can be exploited in logic programs are well known: or-parallelism and and–parallelism. In this work we concentrate on the case of and–parallelism. And-parallelism refers to the parallel execution of the literals in the body of a clause. See, for example, [18] and its references. If only *independent goals* are executed in parallel, both correctness and efficiency can be ensured [23].

5.1 The Annotation Process and Run-time Tests

The annotation (parallelization) process can be viewed as a source–to–source transformation from standard Prolog to a parallel dialect. Herein, we will use the `&` operator [20]. Execution of literals separated by `&` is performed in parallel if sufficient processors are available. Otherwise they will be executed sequentially.

The automatic parallelization process is performed as follows [37]: firstly, if requested by the user, the Prolog program is analyzed using one or more global analyzers. Secondly, since side–effects cannot be allowed to execute freely in parallel, the original program is analyzed using the global analyzer described in [38] which propagates the side–effect characteristics of builtins determining the scope of side–effects. Finally, the *annotators* perform a source–to–source transformation of the program in which each clause is annotated with parallel expressions and conditions which encode the notion of independence used. In doing this they use the information provided by the global analyzers mentioned before.

5.2 An Example: Matrix Multiplication

A Prolog program for matrix multiplication is shown below. The declaration `:-module(mmatrix,[mmultiply/3]).` is used by the (goal dependent) analyzer to determine that only calls to `mmultiply/3` may appear in top-level queries. In this case no information is given about the arguments in calls to the predicate `mmultiply/3` (however, this could be done using one or more `entry` declarations [2]).

```
:-module(mmatrix,[mmultiply/3]).

mmultiply([],_,[]).
mmultiply([V0|Rest], V1, [Result|Others]):-
    multiply(V1,V0,Result), mmultiply(Rest, V1, Others).
multiply([],_,[]).
multiply([V0|Rest], V1, [Result|Others]):-
    vmul(V0,V1,Result), multiply(Rest, V1, Others).
vmul([],[],0).
vmul([H1|T1], [H2|T2], Result):-
    Product is H1*H2, vmul(T1,T2, Newresult),
    Result is Product+Newresult.
```

If, for example, we want to specialize the program for the case in which the first two arguments of `mmultiply/3` are ground values and we inform the analyzer about this, the program would be parallelized without the need for any run-time tests. In our case the analyzer must in principle assume no knowledge regarding the instantiation state of the arguments at the module entry points.

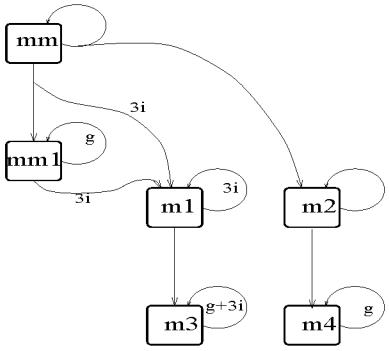

Figure 4: Call Graph of Specialized mmatrix

Figure 3 contains the result of automatic parallelization under these assumptions. `if-then-elses` are written `(cond -> then ; else)`, i.e., using standard Prolog syntax. The predicate `vmul/3` is not shown in Figure 3 because automatic parallelization has not detected any profitable parallelism in it (due to granularity control) and its code remains the same as in the original program.

It is clear from Figure 3 that a good number of run-time tests has been introduced during the parallelization process. If the tests succeed the parallel code is executed. Otherwise the original sequential code is executed. The boolean test `indep(X,Y)` succeeds if and only if `X` and `Y` have no variables in common. For conciseness and efficiency, a series of tests `indep(X1,X2), ..., indep(Xn-1,Xn)` is written as `indep([[X1,X2], ..., [Xn-1,Xn]])`.

Clearly, these tests may cause considerable overhead in run-time performance, to the point of not even knowing at first sight if the parallelized program will offer speedup, i.e., if it will run faster than the sequential one. We will use abstract multiple specialization in order to reduce the run-time overhead and increase the speedup of parallel execution.

It is important to mention that abstract multiple specialization is able to automatically detect and extract some invariants in recursive loops: once a certain run-time test has succeeded it does not need to be checked in the following recursive calls [17]. Figure 4 shows the call graph of the specialized parallel program. The program itself is not shown for space limitations but can be found in [45]. In the figure, `mm` stands for `mmultiply/3` and `m` for `multiply/3`. In the and–or graph computed by analyis there are two or–nodes for predicate `mmultiply/3`, four for `multiply/3`, and eight for `vmul/3`. The minimization algorithm collapses all or–nodes for `vmul/3` into one since the different call patterns do not lead to interesting optimizations. However, two versions are generated for `mm`: `mm` and `mm1` and four for `m`. In Figure 4 edges are labeled with the number of tests which are avoided in each call to the corresponding version with respect to the non specialized program. For example, `g+3i` means that each execution of this specialized version avoids a groundness and three independence tests. It can be seen in the figure that once the groundness test in any of `mm`, `m1`, or `m2` succeeds, it is detected as an invariant, and the more optimized versions `mm1`, `m3`, and `m4` respectively will be used in all remaining iterations.

```
mmultiply([],_,[]).
mmultiply([V0|Rest],V1,[Result|Others]) :-
        (ground(V1), indep([[V0,Rest],[V0,Others],[Rest,Result],[Result,Others]]) ->
             multiply(V1,V0,Result) & mmultiply(Rest,V1,Others)
        ;    multiply(V1,V0,Result), mmultiply(Rest,V1,Others)).

multiply([],_,[]).
multiply([V0|Rest],V1,[Result|Others]) :-
        (ground(V1), indep([[V0,Rest],[V0,Others],[Rest,Result],[Result,Others]]) ->
             vmul(V0,V1,Result) & multiply(Rest,V1,Others)
        ;    vmul(V0,V1,Result), multiply(Rest,V1,Others)).
```

Figure 3: Parallel mmatrix

6. OPTIMIZATION OF DYNAMIC SCHEDULING

Most "second-generation" logic programming languages provide a flexible scheduling in which computation generally proceeds left-to-right, but some calls are dynamically "delayed" until their arguments are sufficiently instantiated. This general form of scheduling, often referred to as *dynamic scheduling*, which can be seen as a (restricted) class of concurrency, increases the expressive power of (constraint) logic programs. Unfortunately, it also has a significant time and space overhead.

In this section we present by means of examples two different classes of transformations. The first class simplifies the delay conditions associated with a particular literal. The second class of transformations reorders a delayed literal and moves it closer to the point where it wakes up. Both classes of transformations essentially preserve the search space and hence the operational behavior of the original program. However, reordering may change the execution order of delayed literals that are woken at exactly the same time. Note that this order is system dependent and it is rare for programmers to rely on a particular ordering.

Using the `CiaoPP` system we have built a tool which automatically optimizes logic programs with delay using the above transformations. Initial experiments suggest that simplification of delay conditions is widely applicable and can significantly speed up execution, while reordering is less applicable but can also lead to substantial performance improvements.

6.1 Programs with Delaying Conditions

In dynamically scheduled languages the execution of some literal can be delayed until a particular delay condition holds. A *delay condition*, *Cond*, takes the current run-time environment and returns *true* or *false* indicating if evaluation can proceed or should be delayed. Typical primitive delay conditions are `ground(X)` and `nonvar(X)`. The latter holds iff `X` is bound to a non-variable term. Delay conditions can be combined to allow more complex delay behaviour. They can be conjoined, written $(Cond_1, Cond_2)$, or disjoined, written $(Cond_1; Cond_2)$.

A *delaying literal* is of the form $when(Cond, L)$, where *Cond* is a delay condition and L is a literal. Evaluation of L will be delayed until *Cond* holds for the current constraint store. Delay information can be *predicate-based* and *literal-based*. In the former, the delaying literal appears as a declaration before the definition of the predicate, each instance of the predicate inheriting the delay condition. In the latter, the delaying literal appears in the body of some clause only affecting the literal L. It is straightforward to use predicate-based declarations to imitate literal-based delay, and vice versa. For simplicity, we will restrict ourselves to literal-based delay.

In logic programs with dynamic scheduling, a *literal* is either an atom or a delaying literal. We are assuming that all rule heads are normalized, since this simplifies the examples and corresponds to what is done in the analyzer.[5] This is not restrictive since programs can always be normalized. However, so as to preserve the behaviour of the original program under dynamic scheduling, the normalization process must ensure that head unifications are performed simultaneously, that is, grouped together in one primitive constraint.

6.2 Simplifying Dynamic Scheduling

Delay conditions may be evaluated each time a variable is touched. Simplifying such conditions can then lead to significant performance improvement. Essentially the behaviour of a delay condition is only relevant during the lifetime of the delaying literal. Hence, we can replace one delay condition by another (more efficient) condition if they are equivalent for all constraint stores that occur during the lifetime of the delaying literal.

EXAMPLE 6.1. *Dynamic scheduling can be used in order to obtain much more general code. Consider for example the following program for naive reverse:*

```
:- module(nrev,[nrev/2]).
nrev([],[]).
nrev([X|Xs], Rs) :- nrev(Xs, R), app(R, [X], Rs).

app([],L,L).
app([X|Xs], Ys, [X|Zs]) :- app(Xs, Ys, Zs).
```

The `nrev/2` predicate can be used in order to reverse a list. For example, the call `nrev([1,2,3],Y)` will return `Y=[3,2,1]`. Since this program does not contain any impurities, we may in principle use it backwards, i.e., a call such as `nrev(X,[1,2,3])` should return `Y=[3,2,1]`. In fact, any Prolog system would compute that. However, if we ask for a second solution, the execution loops! One possible solution to avoid this behaviour is to reorder the two literals in the recursive clause of `nrev/2`, i.e.:
`nrev([X|Xs], Rs) :- app(R, [X], Rs), nrev(Xs, R).`
However, now this program cannot be used forwards. This problem can be solved by means of dynamic scheduling which allows having a definition of `nrev/2` which works in both directions. Such a program is shown below:

[5] `CiaoPP` does not need to normalize programs in order to analyze them, except for programs with dynamic scheduling.

```
nrev([],[]).
nrev([X|Xs], Rs) :-
    when((nonvar(Xs);ground(R)),nrev(Xs, R)),
    when((nonvar(R);nonvar(Rs)),app(R, [X], Rs)).
app([],L,L).
app([X|Xs], Ys, [X|Zs]) :-
    when((nonvar(Xs);nonvar(Zs)),app(Xs, Ys, Zs)).
```

This has the disadvantage that dynamic scheduling may introduce important run-time overhead. However, we can use abstract specialization in order to optimize the above code for the required usage. In fact, our prototype specializer for dynamic scheduling [41] is able to optimize the program back to the original code without delays shown in Example 6.1 if it can infer that at the call the first argument is definitely ground. Also, it will reorder the two literals in the recursive clause of append if analysis guarantees that calls have a free variable in the first argument and the second argument is ground.

6.3 Reordering Delaying Literals

In spite of the apparent simplicity of the specialization of dynamic scheduling, it is indeed rather involved. First, the analysis has to be able to handle logic programs with dynamic scheduling. Doing so accurately is a complex task. Second, the purpose of specialization is not that the final program can be executed without delays but rather that the operational semantics, i.e., the search space, of the program is maintained.

EXAMPLE 6.2. *In order to illustrate this we show the following example in which a naive algorithm for sorting lists is presented. It is based on the specification of the sorting algorithm: the resulting list must be a permutation of the input list and be sorted.*

```
naive_sort(List, Sorted) :-
   when(nonvar(Sorted),sorted_list(Sorted)),
   permute(List, Sorted).

sorted_list([]).
sorted_list([Fst|Oths]) :-
   when(nonvar(Oths),sorted_list1(Fst, Oths)).

sorted_list1(_, []).
sorted_list1(Fst, [Snd|Rest]) :-
   when((ground(Fst),ground(Snd)),Fst =< Snd),
   when(nonvar(Rest),sorted_list1(Snd, Rest)).

permute([],[]).
permute(List,Result):-
   when((nonvar(List);nonvar(Oths)),
                    delete(Elem, List, Oths)),
   Result = [Elem|Perm1],
   permute(Oths, Perm1).

delete(Elem, [Elem|Oths], Oths).
delete(Elem, List, Oths):-
   head(List,Oths) = head([Fst|TM],[Fst|R]),
   when((nonvar(TM);nonvar(R)),delete(Elem, TM, R)).
```

Thanks to the use of dynamic scheduling the code above has the following desirable features: (1) it can be used in order to sort a list; (2) if the second argument is ground, it can be used in order to generate all the possible lists (permutations) of a given sorted list; (3) though it is not a fast sorting algorithm, it behaves relatively well for small lists due to co-routining: generation of the permutation is interleaved with tests of its sortedness as new items are added to the partial solution, i.e., it is a *test while generate* algorithm rather than a *generate and test* one.

Of course, another alternative would have been to write by hand a program which checks sortedness of partial solutions explicitly. This has the disadvantage that it separates the code apart from its specification and that the obvious resulting code is once again not reversible.

EXAMPLE 6.3. *In a call such as* `naive_sort([1,2,3],L)` *the literal* `when(nonvar(Sorted),sorted_list(Sorted))` *will delay at the execution of predicate* `naive_sort/2` *whereas it will definitely not delay after the execution of the literal* `permute(List, Sorted)`. *We may thus be tempted to reorder it across the following literal in the clause, obtaining:*

```
naive_sort(List, Sorted) :-
       permute(List,
Sorted), sorted_list(Sorted).
```

which no longer needs dynamic scheduling. However, this resulting program would definitely be much less efficient than the original one since this changes the co-routining behaviour and thus the search space, and we end up in the generate and test algorithm. □

Though our specializer reordered the literals in the naive reverse example, it does not in this one. This is because the specializer only reorders a delaying literal L_i until after literal L_{i+1} if either (1) L_i is guaranteed not to wake up during the execution of L_{i+i} or (2) if it does, it can only wake up in program points of L_{i+1} which are *final*. More details can be found in [41]. The program obtained by our specializer when the first argument is ground is shown below:

```
naive_sort(List,Sorted) :-
    when(nonvar(Sorted),sorted_list(Sorted)),
    permute(List,Sorted).

sorted_list([]).
sorted_list([Fst|Oths]) :-
    when(nonvar(Oths),sorted_list1(Fst, Oths)).

sorted_list1(_, []).
sorted_list1(Fst, [Snd|Rest]) :-
    when((ground(Fst),ground(Snd)),Fst =< Snd),
    when(nonvar(Rest),sorted_list1(Snd, Rest)).

permute([],[]).
permute(List,Result) :-
    delete(Elem,List,Oths),
    Result=[Elem|Perm1],
    permute(Oths,Perm1).

delete(Elem, [Elem|Oths], Oths).
delete(Elem, List, Oths):-
    head(List,Oths) = head([Fst|TM],[Fst|R]),
    delete(Elem,TM,R).
```

6.4 Automating the Optimization

In order to perform the optimizations discussed, the abstract interpretation framework used has to handle dynamic scheduling. Different analysis frameworks have been proposed for this. In our prototype we use the approach of [8]. For reordering, the analyzer needs to provide, in addition to a description of calling contexts, a description of the set of waking up literals at each program point.

The experimental results in [41] demonstrate that both simplification and reordering can lead to an order of magnitude performance improvement, and that they give reasonable speedups in most benchmarks. This is important

because dynamic scheduling looks set to become increasingly prevalent in (constraint) logic programming languages because of its importance in implementing constraint solvers and controlling search as well as for implementing concurrency. In all these contexts, delay declarations are automatically introduced by the compiler. This has the advantage that it avoids the tedious and error prone task of having to do it by hand. Also, they are a clear target for abstract specialization.

7. INTEGRATION WITH PARTIAL EVALUATION

Most of the practical algorithms for program specialization use, to a greater or lesser degree, information generated by static program analysis. As already mentioned, one of the most widely used techniques for static analysis is abstract interpretation [6]. In fact, some of the relations between abstract interpretation and partial evaluation have been identified before [12, 17, 9, 5, 43, 32, 25, 42, 29, 47, 7].

However, the role of analysis is so fundamental that it is natural to consider whether partial evaluation could be achieved directly by a generic, top-down abstract interpretation system.

7.1 And–Or Graphs Vs. SLD Trees

Almost all existing approaches to the (on-line) partial evaluation of logic programs use the same operational semantics, i.e *SLD resolution*, for both program execution and partial evaluation. Different alternative derivations of SLD resolution which may occur during execution constitute different branches in the *SLD tree*. See for example [34]. In partial deduction a slight modification to this semantics is required in order to allow incomplete derivations and thus incomplete SLD trees.

However, it is known [32] that the propagation of success information during partial evaluation is not optimal compared to that potentially achievable by abstract interpretation. The higher accuracy of abstract interpretation has already been hinted in Example 4.2.

We now show a further example of the power of abstract interpretation. This time, rather than the concrete domain we will use the abstract domain *eterms* [50] currently implemented in the CiaoPP system, and which is based on the concept of regular types [13]. Note that in this example the concrete domain cannot be used straight away, since the set of values which need to be represented is infinite.

EXAMPLE 7.1. *Consider the following program and the initial call* `r(X)`

```
r(X) :- q(X),p(X).
q(a).
q(f(X)) :- q(X).
p(a).
p(f(X)) :- p(X).
p(g(X)) :- p(X).
```

It can be observed that the third clause for `p` *can be eliminated in the specialized program, since the call substitution for* `p(X)` *(i.e., the success substitution for* `q(X)`*) is of the form* `X=a` *or* `X=f(a)` *or* `X=f(...f(a)...)`*. Thus, the clause* `p(g(X)) :- p(X).` *is useless. Our implementation of the abstract domain* eterms *is able to determine that the value of* `X` *in any call to* `p(X)` *is described by the regular type* `rt` *whose definition as a regular unary Prolog program follows:*

```
rt(a).
rt(f(A)) :- rt(A).
```

Our specializer is in fact able to use this information in order to remove the useless clause mentioned above. Note that standard partial evaluation algorithms based on unfolding will not be able to eliminate the third clause for `p`, since an atom of the form `p(X)` will be produced, no matter what local and global control is used.[6]

In addition to allowing the elimination of useless clauses, our specialization system is able to perform more aggressive optimizations, as shown in the example below.

EXAMPLE 7.2. *Consider the following program which defines a predicate* `flatten_and_sort/2`.

```
flatten_and_sort(Struct,Sorted_List):-
        sorted_int_list(Struct),
        Sorted_List=Struct.
flatten_and_sort(Struct,Sorted_List):-
        int_list(Struct),
        sort(Struct,Sorted_List).
flatten_and_sort(Struct,Sorted_List):-
        list_of_int_lists(Struct),
        flatten_list(Struct,Unsorted_List),
        sort(Unsorted_List,Sorted_List).
flatten_and_sort(Struct,Sorted_List):-
        tree(Struct),
        flatten_tree(Struct,Unsorted_List),
        sort(Unsorted_List,Sorted_List).
```

The argument `Struct` is a data structure which can be: a sorted list of integers, a list of integers, a list of lists of integers, or a tree which stores an integer in each non-leaf node. The predicate first determines which of the four possibilities mentioned above is the case and then, if needed, it uses the appropriate procedure for flattening before sorting the list of arguments, which is the output of the procedure. Clearly, if the input data structure is a list of integers there is no need for flattening the list. Furthermore, if it is already sorted, there is no need to sort it either. Though we could define a flatten predicate which is able to flatten both lists and binary trees, it is often the case that distinct predicates for flattening lists and for flattening trees already exist (in different libraries).

We show below the Prolog definition of the properties `sorted_int_list/1`, `int_list/1`, and `list_of_int_lists/1`. It can be observed that the last two predicates are indeed unary logic programs which correspond to deterministic regular types. This is indicated to CiaoPP with the declaration `regtype`.

```
sorted_int_list([]).
sorted_int_list([N]):-  int(N).
sorted_int_list([A,B|R]):- int(A), int(B),
        A =< B, sorted_int_list([B|R]).

:- regtype int_list/1.
int_list([]).
int_list([H|L]):- int(H), int_list(L).

:- regtype list_of_int_lists/1.
list_of_int_lists([]).
list_of_int_lists([H|L]):-
        int_list(H), list_of_int_lists(L).

:- regtype tree/1.
tree(void).
tree(t(L,N,R)):- int(N), tree(L), tree(R).
```

[6]Conjunctive partial deduction [33] can solve this problem in a completely different way.

The `regtype` declaration is checked by `CiaoPP` against the code defining the property. If the code does not correspond to a deterministic regular type, an error message is issued. If it is, this information can be used by the specializer in order to be able to abstractly execute to the value true the whole execution of the predicate. The sufficient conditions for this are (1) the predicate does not perform any side-effects, and (2) the calling abstract substitution must be equal or more particular than the success substitution for the predicate. Note that abstractly executing a predicate call to false using regular types does not need the `regtype` declaration. Any call to a predicate p can be abstractly executed to false if (1) execution of p is guaranteed not to perform any side-effects (2) the call substitution is incompatible with the success substitution of p or equivalently, the success substitution using goal-dependent analysis for p and λ_p^c is the empty substitution $\bot$. This is further discussed in Section 7.3. For example, if we call `sorted_int_list(Struct)` with `Struct` bound to a binary tree, the system can determine that this call is incompatible with the success type of `sorted_int_list`, which for the regular type analysis is approximated by `int_list`.

For example, the above program when specialized using the *eterms* domain for the call `main/0`, defined as:

```
main:-int_list(L),append(L,[3],L1),flatten_and_sort(L1,_).
```

optimizes the definitions of `flatten_and_sort/2` and `int_list/2` as shown below.

```
flatten_and_sort(Struct,Sorted_List) :-
      sorted_int_list(Struct), sort(Struct,Sorted_List).
flatten_and_sort(Struct,Sorted_List) :-
      sort(Struct,Sorted_List).
sorted_int_list([]).
sorted_int_list([N]).
sorted_int_list([A,B|R]) :- A=<B, sorted_int_list([B|R]).
```

Since analysis using *eterms* infers that the call to `flatten_and_sort/2` has got a non-empty list of integers as first argument, the specializer is able to abstractly execute the tests for `list_of_int_lists/1` and `tree/1` to false, since they are incompatible with their calling types. In addition, the `list/1` test in the second clause for `flatten_and_sort/2` has been abstractly executed to true, the same as the `integer/1` tests in `sorted_int_list/1`. This is an example in which abstract execution allows "executing" at compile-time a test whose execution would require traversing the data structure at run-time.

The examples above show that and–or graphs allow a level of success information propagation not possible in traditional partial evaluation. This observation already provides motivation for studying the integration of full partial evaluation in an analysis/specialization framework based on abstract interpretation.

7.2 Partial Evaluation using And–Or Graphs

We now discuss how the global and local control aspects of on-line partial evaluation appear in the setting of abstract interpretation algorithms.

7.2.1 Global Control in Abstract Interpretation

Effectiveness of traditional partial deduction greatly depends on the set of atoms $\mathbf{A} = \{A_1, \ldots, A_n\}$ for which (specialized) code is to be generated. This set is mainly determined by the global control used. However, in abstract specialization the role of the atoms in $\mathbf{A}$ is played by the set of or–nodes $Analysis(P, p, \lambda, D_\alpha)$. The choice of abstract domain and widening operators (if any) will determine the number of or–nodes (equivalently, $\mathbf{A}$). The finer-grained the abstract domain is, the larger the set $\mathbf{A}$ will be. In conclusion, the role of so-called global control in partial evaluation is played in abstract interpretation by our particular choice of abstract domain and widening operators (which are strictly required for ensuring termination when the abstract domain contains ascending chains which are infinite – as is the case for the concrete domain and for domains based on regular types).

Note that the specialization framework we propose is very general. Depending on the kind of optimizations we are interested in performing, different domains (and widening operators) should be used and thus different $\mathbf{A}$ sets would be obtained.

7.2.2 Local Control in Abstract Interpretation

Local control in partial evaluation determines how each atom in $\mathbf{A}$ should be unfolded. However, in traditional abstract interpretation frameworks each or–node is related by just one (abstract) unfolding step to its children. This corresponds to a trivial local control (unfolding rule) in partial evaluation.

Note that if we use abstract domains for analysis which allow propagating enough information about the success of an or–node, it is possible to perform useful specialization on other or-nodes. This requires that the *lub* operator not lose "much" information, for example by allowing sets of abstract substitutions. The advantage of this method is that no modification of the abstract interpretation framework is required. An example of this has been shown in Example 7.1. Such specialization is not possible by methods based on unfolding (unfolding is a standard program transformation technique in which an atom in the body of a clause, i.e., a call to a procedure, is conceptually replaced by the code of such procedure).

Another approach to overcoming this limitation of abstract interpretation is the use of *node-unfolding* [47]. Node-unfolding is a *graph* transformation technique which given an and–or graph AO and an or–node N in AO builds a new and–or graph AO'. Such graph transformation mimics the effect of traditional unfolding.

EXAMPLE 7.3. *Consider the program below. The analysis graph generated without performing any node-unfolding is shown in Figure 5 as AO, using the concrete domain as abstract domain and the* most specific generalization *(msg) as lub operator for summarizing different success substitutions into one. As discussed in Section 7.2.3 below, the msg is a rather crude lub operator. However, we use it for the sake of clarity of the example.*

```
p(X):- q(X), r(X).
q(a).
q(b).
r(a).
r(b).
```

AO' is an analysis graph for the same program but this time the or–node $\langle q(X), \{\}, \{\}\rangle$ has been unfolded. Finally, graph AO'' in the figure is the result of applying node-unfolding twice to AO', once w.r.t. $\langle p(a), \{\}, \{\}\rangle$ and another one w.r.t $\langle p(b), \{\}, \{\}\rangle$. The code generated by $code\_gen(AO'')$ is the program:

```
p(a).
p(b).
```

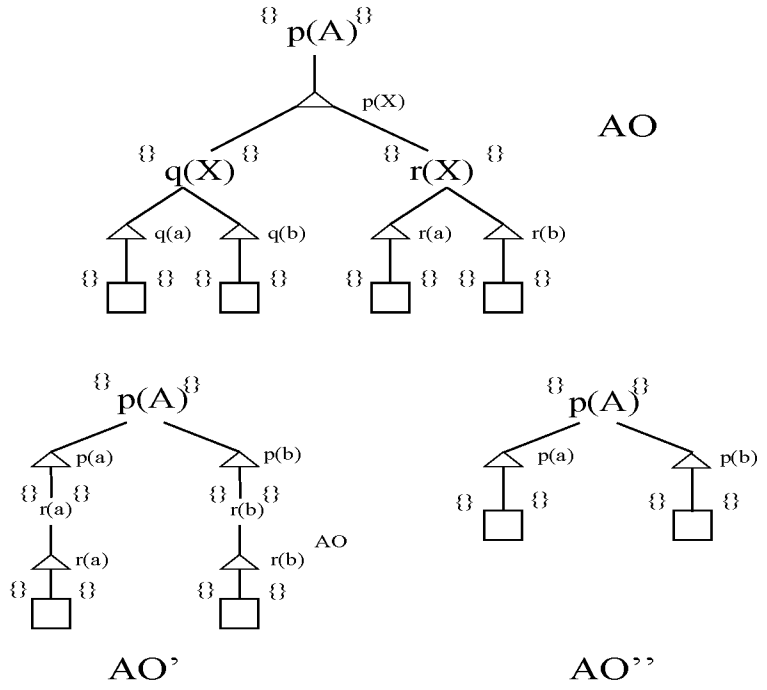

Figure 5: Example Node Unfoldings

An important question is the moment at which node-unfolding is performed, i.e., during or after building *AO*. The simplest possibility is to perform node-unfolding of an or–node prior to computing its success substitution. This corresponds to what is done in partial deduction: local control is performed first and then atoms are passed to global control. It allows performing node-unfolding after computing the success-substitution of an or-node, even after computing the final and–or graph. This allows having more information prior to deciding whether to unfold a node or not. Thus, we consider it a more challenging approach. The main difficulty lies in being able to efficiently rebuild the analysis and–or graph so as to reach a fixpoint after the graph is modified by node-unfolding. We believe that this cost can be kept quite reasonable by the use of incremental analysis techniques such as those presented in [22, 44].

7.2.3 *Abstract Domains and Widenings for Partial Evaluation*

We now address the features which an abstract domain (and associated widening operators) should have in order to be appropriate for performing partial evaluation within the abstract specialization framework. They should (1) simulate the effect of unfolding, which is how bindings are propagated in partial evaluation. The abstract domain has to be capable of tracking such bindings. This suggests that domains based on term structure are required. In addition, the domain (2) needs to capture *disjunctive information*. This makes it possible to distinguish, in a single abstract substitution, several bindings resulting from different branches of computation. A term domain whose least upper bound is based on the *msg* (most specific generalization), for instance, will rapidly lose information about multiple answers since all substitutions are combined into one binding.

We now discuss two classes of domains which have the above mentioned features. One class is based on sets of depth-k substitutions with set union as the least upper bound operator. However, uniform depth bounds are usually either too imprecise (if k is too small) or generate much redundancy if larger values of k are chosen. One way to eliminate the depth-bound k in the abstract domain is to depend on a suitable widening operator which will guarantee that the set of or–nodes remains finite. Many techniques have been developed for global control of partial evaluation. Such techniques make use of advanced data structures such as *characteristic trees* [11], [27] (related to *neighborhoods* [49]), *trace-terms* [14], and *global trees* [36], and combinations of them [31]. Thus, it seems possible to adapt these techniques to the case of abstract interpretation and formalize them as widening operators.

The second class of domains are those based on regular-types [13, 19, 50] and seem very good candidates, their main drawback being that inter-argument dependencies are lost. Independently of our work in `CiaoPP`, recently there has been a lot of interest in the application of regular types for improving partial evaluation [15, 30]. The use of non-deterministic regular types [16] presents an interesting trade-off since on one hand they allow improved accuracy but on the other they require a higher computational cost and their applicability to program specialization should be further explored.

7.3 Code Generation using Success Substitutions

One important feature of abstract specialization not available in partial evaluation is that for each or-node, in addition to a call substitution, there is also an abstract substitution which describes the success of the call. If the properties

captured by the abstract domain are downwards closed (as is the case with variable bindings), it is natural to consider specialization w.r.t. success substitutions rather than call substitutions (only). We first recall some notation from [47].

DEFINITION 7.4 (PARTIAL CONCRETIZATION). *A function* $part\_conc : D_\alpha \rightarrow D$ *is a* partial concretization *iff* $\forall \lambda \in D_\alpha \; \forall \theta' \in \gamma(\lambda) \; \exists \theta''$ *s.t.* $\theta' = part\_conc(\lambda)\theta''$.

$part\_conc(\lambda)$ can be regarded as containing (part of) the definite information about concrete bindings that the abstract substitution λ captures. Note that different partial concretizations of an abstract substitution λ with different accuracy may be considered. For example if the abstract domain is a depth-k abstraction and $\lambda = \{X/f(f(Y)) or X/f(a)\}$, a most accurate $part\_conc(\lambda)$ is $\{X/f(Z)\}$. Note also that $part\_conc(\lambda) = \epsilon$ where ϵ is the empty substitution, is a trivially correct partial concretization of any λ.

It is straightforward to modify Algorithm 4.1 in order to exploit answer substitutions as well. Such algorithm can be found in [47]. Specialization w.r.t. answers will in general provide further specialized (and optimized) programs as in general the success substitution (which describes answers) computed by abstract interpretation is more informative (restricted) than the call substitution. However, this cannot be done for example if the program contains calls to extra-logical predicates such as `var/1`.

Specializing w.r.t answer substitutions enables optimizations which are not possible to achieve by finite unfolding. For example, abstract interpretation can detect both finite and infinite failure of a predicate p. In both cases, the abstract success substitution for p will be $\bot$. If p does not perform side effects, the definition of p generated by our specializer is $\mathtt{p}(\overline{t})\mathtt{:-\ fail.}$, as it is known to produce no answers. Even if the success substitution λ^s for $\langle p, \lambda^c, \lambda^s \rangle$ is not $\bot$, individual clauses for p whose success substitution is $\bot$ (useless clauses) for the considered λ^c are removed from the final program.

Note that the specialized program may fail finitely while the original one loops. We believe this kind of optimizations are desirable in most cases. However, optimization w.r.t. answers is optional in our system.

8. RELATED WORK

Abstract specialization is a framework which can be used successfully in different contexts. We have discussed its application to program parallelization and optimization of dynamic scheduling. The framework is generic in that it can be instantiated with different abstract domains which provide different kinds of information according to the optimizations which we aim at performing. If the abstract domain captures term structure then it is possible to obtain information which can then be used to perform optimizations which are very related to those which take place during partial evaluation.

The integration of partial evaluation and abstract interpretation has been attempted before, both from the partial evaluation and the abstract interpretation perspectives. Some preliminary studies are [12, 9] in which an integration is attempted from the point of view of partial evaluation. Another integration in the context of functional programs is presented in [5]. On the other hand, the drawbacks of traditional partial evaluation techniques for propagating success information are identified in [32] and some of the possible advantages of a full integration of partial evaluation and abstract interpretation are presented in [25].

From an abstract interpretation perspective, the integration has also received considerable attention. The first complete framework for multiple specialization based on abstract interpretation is presented in [51]. The first implementation and experimental evaluation appears in [43]. However, these systems do not perform unfolding.

To the best of our knowledge, the first relatively satisfactory framework for the integration of abstract interpretation and partial evaluation is [42, 47].

A completely different framework for the integration of partial deduction and abstract interpretation is presented in [29]. In this formulation a top-down specialization algorithm is presented which assumes the existence of an *abstract unfolding* and an *abstract resolution* operation and which generalizes existing algorithms for partial evaluation. Such framework provides interesting insights on the problems involved together with correctness conditions which can be used to prove that a given specialization framework, which possibly uses abstract interpretation, is correct. One important difference is that in our approach a single (and already existing) top-down abstract interpretation algorithm augmented with an unfolding rule performs propagation of both the call and success patterns in an integrated fashion, whereas in [29] the success propagation used is added in an ad hoc way and is not multivariant, and thus less precise.

Another difference between the two approaches is that [29] is capable of dealing with conjunctions and not only atoms.

The need for more general information than the concrete substitutions handled by partial evaluation has been identified repeatedly in previous work, such as [5, 39]. Though the aims of abstract specialization and those of [39] are quite similar, the means proposed to achieve them are completely different. Also, abstract interpretation is not used and it sticks to the more traditional SLD semantics.

More recently, [7] presents a very general view which integrates program transformation and abstract interpretation. This result allows formalizing partial evaluation as an abstract interpretation (as done by abstract specialization). This new formalization of program transformation may enable other novel program optimization techniques.

9. CONCLUSIONS

Abstract specialization can be seen as a semantic approach much in the same way as existing frameworks for partial deduction [35, 26, 10, 28] and also as other attempts at the integration of partial evaluation and abstract interpretation of logic programs [29, 15, 30]. One of the main differences between abstract specialization and the aforementioned techniques is the underlying semantics. Abstract specialization is based on and–or trees whereas the rest are based on SLD trees. Though SLD-trees have the conceptual advantage that the semantics used for program specialization is almost identical to that used during program execution, our approach has other practical and conceptual advantages. For example, optimizations based on and–or trees can be done to preserve number and order of solutions, an issue often overlooked by traditional partial deduction systems. Furthermore, they allow performing optimizations not achievable by means of unfolding, including the detection of infinite failure.

A pragmatic motivation for this work is the availability of off-the-shelf generic abstract interpretation engines such as the one in CiaoPP [21][7] which greatly facilitate the efficient implementation of analyses. Such analysis can deal with all features of real programs [2] in an accurate way, including builtins, libraries and modules [46]. But, more generally, we argue that *the existence of such an abstract interpreter in advanced optimizing compilers is likely*, and thus using the analyzer itself to perform partial evaluation can result in a great simplification of the architecture of the compiler.

10. REFERENCES

[1] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[2] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

[3] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[4] C. Consel and O. Danvy. Tutorial Notes on Partial Evaluation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL'93*, pages 493–501, 1993. ACM.

[5] C. Consel and S. Koo. Parameterized partial deduction. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, July 1993.

[6] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[7] P. Cousot and R. Cousot. Systematic Design of Program Transformation Frameworks by Abstract Interpretation. In *POPL'02: 29ST ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–190, 2002. ACM.

[8] M. G. de la Banda, K. Marriott, and P. Stuckey. Efficient Analysis of Constraint Logic Programs with Dynamic Scheduling. In *International Logic Programming Symposium*, 1995. MITPress.

[9] J. Gallagher. Static Analysis for Logic Program Specialization. In *Workshop on Static Analysis WSA'92*, pages 285–294, 1992.

[10] J. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of ACM PEPM'93*, pages 88–98. ACM Press, 1993.

[11] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(1991):305–333, 1991.

[12] J. Gallagher, M. Codish, and E. Shapiro. Specialisation of Prolog and FCP Programs Using Abstract Interpretation. *New Generation Computing*, 6(2–3):159–186, 1988.

[13] J. Gallagher and D. de Waal. Fast and precise regular approximations of logic programs. In P. Van Hentenryck, editor, *International Conference on Logic Programming*. MIT Press, 1994.

[14] J. Gallagher and L. Lafave. Regular approximation of computation paths in logic and functional languages. In *Partial Evaluation*, volume 1110, pages 115 – 136. Springer Verlag LNCS, 1996.

[15] J. Gallagher and J. Peralta. Using regular approximations for generalisation during partial evaluation. In *Proc. ACM PEPM*, pages 44–51. ACM Press, 2000.

[16] J. Gallagher and G. Puebla. Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In *Int. Symp. on Practical Aspects of Declarative Languages*, number 2257 in LNCS. Springer-Verlag, 2002.

[17] F. Giannotti and M. Hermenegildo. A Technique for Recursive Invariance Detection and Selective Program Specialization. In *Proc. PLILP*, number 528 in LNCS, pages 323–335. Springer-Verlag, August 1991.

[18] G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: a Survey. *ACM TOPLAS*, 23(4):472–602, July 2001.

[19] P. V. Hentenryck, A. Cortesi, and B. L. Charlier. Type analysis of prolog using type graphs. *Journal of Logic Programming*, 22(3):179 – 210, 1994.

[20] M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[21] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *SAS'03*, LNCS. Springer-Verlag, June 2003.

[22] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM TOPLAS*, 22(2):187–223, March 2000.

[23] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.

[24] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prenctice Hall, New York, 1993.

[25] N. D. Jones. Combining Abstract Interpretation and Partial Evaluation. In *SAS*, number 1140 in LNCS, pages 396–405. Springer-Verlag, 1997.

[7] More information on Ciao and CiaoPP is available at www.clip.dia.fi.upm.es

[26] J. Komorovski. An Introduction to Partial Deduction. In *Meta Programming in Logic, Proceedings of META'92*, volume 649 of *LNCS*, pages 49–69. Springer-Verlag, 1992.

[27] M. Leuschel. Ecological partial deduction: Preserving characteristic trees without constraints. In M. Proietti, editor, *Proc. LOPSTR*. Springer-Verlag, 1995.

[28] M. Leuschel. *Advanced Techniques for Logic Program Specialisation*. PhD thesis, K.U. Leuven, May 1997.

[29] M. Leuschel. Program Specialisation and Abstract Interpretation Reconciled. In *Joint Int. Conference and Symposium on Logic Programming*, June 1998.

[30] M. Leuschel and S. Gruner. Abstract conjunctive partial deduction using regular types and its application to model checking. In *Logic Program Synthesis and Transformation (LOPSTR)*, number 2372 in LNCS. Springer, 2001.

[31] M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 263–283, Schloß Dagstuhl, 1996.

[32] M. Leuschel and D. Schreye. Logic program specialisation: How to be more specific. In H. Kuchen and S. Swierstra, editors, *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, LNCS 1140, pages 137–151, Aachen, Germany, September 1996.

[33] M. Leuschel, D. D. Schreye, and D. A. de Waal. A conceptual embedding of folding into partial deduction: towards a maximal integration. In M. Maher, editor, *Proceedings of the Joint Int,. Conf. and Symp. on Logic Programming (JICSLP'96)*. MIT Press, 1996.

[34] J. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.

[35] J. Lloyd and J. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11(3–4):217–242, 1991.

[36] B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–611, Shonan Village Center, Japan, June 1995. MIT Press.

[37] K. Muthukumar, F. Bueno, M. G. de la Banda, and M. Hermenegildo. Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism. *Journal of Logic Programming*, 38(2):165–218, February 1999.

[38] K. Muthukumar and M. Hermenegildo. Complete and Efficient Methods for Supporting Side Effects in Independent/Restricted And-parallelism. In *1989 International Conference on Logic Programming*, pages 80–101. MIT Press, June 1989.

[39] A. Pettorossi and M. Proietti. A theory of logic program specialization and generalization for dealing with input data properties. In Springer-Verlag, editor, *Dagstuhl Seminar on Partial Evaluation*, number 1110 in LNCS, 1996.

[40] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[41] G. Puebla, M. G. de la Banda, K. Marriott, and P. Stuckey. Optimization of Logic Programs with Dynamic Scheduling. In *1997 International Conference on Logic Programming*, pages 93–107, Cambridge, MA, June 1997. MIT Press.

[42] G. Puebla, J. Gallagher, and M. Hermenegildo. Towards Integrating Partial Evaluation in a Specialization Framework based on Generic Abstract Interpretation. In M. Leuschel, editor, *Proceedings of the ILPS'97 Workshop on Specialization of Declarative Programs*, October 1997. Post ILPS'97 Workshop.

[43] G. Puebla and M. Hermenegildo. Implementation of Multiple Specialization in Logic Programs. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 77–87. ACM Press, June 1995.

[44] G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium*, number 1145 in LNCS, pages 270–284. Springer-Verlag, September 1996.

[45] G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999.

[46] G. Puebla and M. Hermenegildo. Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs. In *Special Issue on Optimization and Implementation of Declarative Programming Languages*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.

[47] G. Puebla, M. Hermenegildo, and J. Gallagher. An Integration of Partial Evaluation in a Generic Abstract Interpretation Framework. In O. Danvy, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, number NS-99-1 in BRISC Series, pages 75–85. University of Aarhus, Denmark, January 1999.

[48] H. Tamaki and M. Sato. Unfold/Fold Transformations of Logic Programs. In *Second International Conference on Logic Programming*, pages 127–138, Uppsala, Sweden, 1984.

[49] V. Turchin. The algorithm of generalization in the supercompiler. In D. B. rner, A. Ershov, and N. Jones, editors, *Proc. of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, pages 531–549. North-Holland, 1988.

[50] C. Vaucheret and F. Bueno. More precise yet efficient type inference for logic programs. In *International Static Analysis Symposium*, number 2477 in LNCS, pages 102–116. Springer-Verlag, September 2002.

[51] W. Winsborough. Multiple Specialization using Minimal-Function Graph Semantics. *Journal of Logic Programming*, 13(2 and 3):259–290, July 1992.